Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks

Jiaqing Du*, Sameh Elnikety†, Amitabha Roy*, Willy Zwaenepoel*
*EPFL †Microsoft Research

Abstract

We propose two protocols that provide scalable causal consistency for both partitioned and replicated data stores using dependency matrices (DM) and physical clocks. The DM protocol supports basic read and update operations and uses two-dimensional dependency matrices to track dependencies in a client session. It utilizes the transitivity of causality and sparse matrix encoding to keep dependency metadata small and bounded. The DM-Clock protocol extends the DM protocol to support read-only transactions using loosely synchronized physical clocks.

We implement the two protocols in Orbe, a distributed key-value store, and evaluate them experimentally. Orbe scales out well, incurs relatively small overhead over an eventually consistent key-value store, and outperforms an existing system that uses explicit dependency tracking to provide scalable causal consistency.

1 Introduction

Distributed data stores are a critical infrastructure component of many online services. Choosing a consistency model for such data stores is difficult. The CAP theorem [7, 10] shows that among Consistency, Availability, and (network) Partition-tolerance, a replicated system can only have two properties out of the three.

A strong consistency model, such as *linearizability* [11] and *sequential consistency* [15], does not allow high availability under network partitions. In contrast, *eventual consistency* [24], a weak model, provides high availability and partition-tolerance, as well as low update latency. It guarantees that replicas eventually converge to the same state provided no updates take place for a long time. However, it does not guarantee any order on applying replicated updates. *Causal consistency* [2] is weaker than sequential consistency but stronger than eventual consistency. It guarantees that replicated updates are applied at each replica in an order that preserves causality [2, 14] while providing availability under network partitions. Furthermore, client operations have low latency because they are executed at a local replica and do not require coordination with other replicas.

The problem addressed in this paper is providing a scalable and efficient implementation of causal consistency for both partitioned and replicated data stores. Most existing causally consistent systems [13, 19, 22] adopt variants of *version vectors* [2, 21], which are designed for purely replicated data stores. Version vectors do not scale when partitioning is added to support a data set that is too large to fit on a single server. They still view all partitions as one logical replica and require a single serialization point across partitions for replication, which limits the replication throughput [17]. A scalable solution should allow replicas of different partitions to exchange updates in parallel without serializing them at a centralized component.

COPS [17] identifies this problem and provides a solution that explicitly tracks causal dependencies at the client side. A client stores every accessed item as dependency metadata and associates this metadata with each update operation issued to the data store. When an update is propagated from one replica to another for replication, it carries the dependency metadata. The update is applied at the remote replica only when all its dependencies are satisfied at that replica. COPS provides good scalability. However, tracking every accessed item explicitly can lead to large dependency metadata, which increases storage and communication overhead and affects throughput. Although COPS employs a number of techniques to reduce the size of dependency metadata, it does not fundamentally solve the problem. When supporting causally consistent read-only transactions, the

SOCC'13, October 01 - 03 2013, Santa Clara, CA, USA
Copyright 2013 ACM 978-1-4503-2428-1/13/10$15.00.
http://dx.doi.org/10.1145/2523616.2523628

dependency metadata overhead is still high under many workloads.

In this paper, we present two protocols and one optimization that provide a scalable and efficient implementation of causal consistency for both partitioned and replicated data stores.

The first protocol uses two-dimensional *dependency matrices* (DMs) to compactly track dependencies at the client side. We call it *the DM protocol*. This protocol supports basic read and update operations. It associates with each update the dependency matrix of its client session. Each element in a dependency matrix is a scalar value that represents all dependencies from the corresponding data store server. The size of dependency matrices is bounded by the total number of servers in the system. Furthermore, the DM protocol resets the dependency matrix of a client session after each update operation, because prior dependencies need not to be tracked due to the transitivity of causality. With sparse matrix encoding, the DM protocol keeps the dependency metadata of each update small.

The second protocol extends the DM protocol to support causally consistent read-only transactions by using loosely synchronized physical clocks. We call it *the DM-Clock protocol*. In addition to the dependency metadata required by the DM protocol, this protocol assigns to each state an update timestamp obtained from a local physical clock and guarantees that the update timestamp order of causally related states is consistent their causal order. With this property, the DM-Clock protocol provides causally consistent snapshots of the data store to read-only transactions by assigning them a snapshot timestamp, which is also obtained from a local physical clock.

We also propose *dependency cleaning*, an optimization that further reduces the size of dependency metadata in the DM and DM-Clock protocols. It is based on the observation that once a state and its dependencies are fully replicated, any subsequent read on the state does not introduce new dependencies to the client session. More messages need to be exchanged, however, for a server to be able to decide that a state is fully replicated, which leads to a tradeoff which we study. Dependency cleaning is a general technique and can be applied to other causally consistent systems.

We implement the two protocols and dependency cleaning in *Orbe*, a distributed key-value store, and evaluate them experimentally. Our evaluation shows that Orbe scales out as the number of data partitions increases. Compared with an eventually consistent system, it incurs relatively little performance overhead for a large spectrum of workloads. It outperforms COPS under many workloads when supporting read-only transactions.

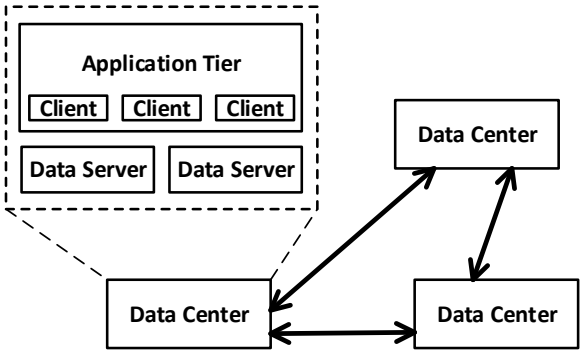

Figure 1: System architecture. The data set is replicated by multiple data centers. Clients are collocated with the data store in the data center and are used by the application tier to access the data store.

In this paper, we make the following contributions:

- The DM protocol that provides scalable causal consistency and uses dependency matrices to keep the size of dependency metadata under control.
- The DM-Clock protocol that provides read-only transactions with causal snapshots using loosely synchronized physical clocks by extending the DM protocol.
- The dependency cleaning optimization that reduces the size of dependency metadata.
- An implementation of the above protocols and optimization in Orbe as well as an extensive performance evaluation.

2 Model and Definition

In this section we describe our system model and define causality and causal consistency.

2.1 Architecture

We assume a distributed key-value store that manages a large set of data items. The key-value store provides two basic operations to the clients:

- PUT(key, val): A PUT operation assigns value *val* to an item identified by *key*. If item *key* does not exist, the system creates a new item with initial value *val*. If *key* exists, a new version storing *val* is created.
- val $\leftarrow$ GET(key): The GET operation returns the value of the item identified by *key*.

An additional operation that provides read-only transactions is introduced later in Section 4.

The data store is partitioned into N partitions, and each partition is replicated by M replicas. A data item is assigned to a partition based on the hash value of its key.

In a typical configuration, as shown in Figure 1, the data store is replicated at M different data centers for high availability and low operation latency. The data store is fully replicated. All N partitions are present at each data center.

The application tier relies on the clients to access the underlying data store. A client is collocated with the data store servers in a particular data center and only accesses those servers in the same data center. A client does not issue the next operation until it receives the reply to the current one. Each operation happens in the context of a client session. A client session maintains a small amount of metadata that tracks the dependencies of the session.

2.2 Causal Consistency

Causality is a *happens-before* relationship between two events [2, 14]. We denote causal order by $\rightsquigarrow$. For two operations a and b, if $a \rightsquigarrow b$, we say b depends on a or a is a dependency of b. $a \rightsquigarrow b$ if and only if one of the following three rules holds:

- Thread-of-execution. a and b are in a single thread of execution. a happens before b.
- Reads-from. a is a write operation and b is a read operation. b reads the state created by a.
- Transitivity. There is some other operation c that $a \rightsquigarrow c$ and $c \rightsquigarrow b$.

We define the *nearest dependencies* of a state as all the states that it directly depends on, without relying on the transitivity of causality.

To provide causal consistency when replicating updates, a replica does not apply an update propagated from another replica until all its causal dependency states are installed locally.

3 DM Protocol

In this section, we present the DM protocol that provides scalable causal consistency using dependency matrices. This protocol is scalable because replicas of different partitions exchange updates for replication in parallel, without requiring a global serialization point.

Dependency matrices are, for systems that support both replication and partitioning, the natural extension of version vectors, for systems that support only replication. A row of a dependency matrix is a version vector that stores the dependencies from replicas of a partition.

A client stores the nearest dependencies of its session in a dependency matrix and associates it with each update request to the data store. After a partition at the client's local data center executes the update, it propagates the update with its dependency matrix to the replicas of that partition at remote data centers. Using the dependency matrix of the received update, a remote replica waits to apply the update until all partitions at its data center store the dependency states of the update.

Symbols	Definitions
N	number of partitions
M	number of replicas per partition
c	a client
DM_c	dependency matrix of c, $N \times M$ elements
PDT_c	physical dependency timestamp of c
p_n^m	a server that runs m^{th} replica of n^{th} partition
VV_n^m	(logical) version vector of p_n^m, M elements
PVV_n^m	physical version vector of p_n^m, M elements
$Clock_n^m$	current physical clock time of p_n^m
d	an item, tuple $\langle k, v, ut, put^1, dm, rid \rangle$
k	item key
v	item value
ut	(logical) update timestamp
put	physical update timestamp
dm	dependency matrix, $N \times M$ elements
rid	source replica id
t	a read-only transaction, tuple $\langle st, rs \rangle$
st	(physical) snapshot timestamp
rs	readset, a set of read items

Table 1: Definition of symbols.

3.1 Definitions

The DM protocol introduces dependency tracking data structures at both the client and server side. It also associates dependency metadata for each item. Table 1 provides a summary of the symbols used in the protocol. We explain their meanings in details below.

Client States. Without losing generality, we assume a client has one session to the data store. A client c maintains for its session a dependency matrix, DM_c, which consists of $N \times M$ non-negative integer elements. DM_c tracks the nearest dependencies of a client session. $DM_c[n][m]$ indicates that the client session potentially depends on the first $DM_c[n][m]$ updates at partition p_n^m, the mth replica of the nth partition.

Server States. Each partition maintains a *version vector* (VV) [2, 21]. The version vector of partition p_n^m is VV_n^m, which consists of M non-negative integer elements. $VV_n^m[m]$ counts the number of updates p_n^m has executed locally. $VV_n^m[i]$ $(i \neq m)$ indicates that p_n^m has applied the first $VV_n^m[i]$ updates propagated from p_n^i, a replica of the same partition.

A partition updates an item by either executing an update request from its clients or by applying a propagated update from one of its replicas at other data centers. We call the partition that updates an item to the current value by executing a client request the *source partition* of the item.

Item Metadata. We represent an item d as a tuple

[1] *put* is only used in the DM-Clock protocol.

$\langle k, v, ut, dm, rid \rangle$. k is a unique key that identifies the item. v is the value of the item. ut is the *update timestamp*, the logical creation time of the item at its source partition. dm is the dependency matrix, which consists of $N \times M$ non-negative integer elements. $dm[n][m]$ indicates that d potentially depends on the first $dm[n][m]$ updates at partition p_n^m, a prefix of its update history. *rid* is the *source replica id*, the replica id of the item's source partition.

We use sparse matrix encoding to encode dependency matrices. Zero elements in a dependency matrix do not use any bits after encoding. Only non-zero elements contribute to the actual size.

3.2 Protocol

We now describe how the DM protocol executes GET and PUT operations and replicates PUTs.

GET. Client c sends a request $\langle \text{GET } k \rangle$ to a partition at the local data center, where k is the key of the item to read. Upon receiving the request, partition p_n^m obtains the read item, d, and sends a reply $\langle \text{GETREPLY } v_d, ut_d, rid_d \rangle$ back to the client. Upon receiving the reply, the client updates its dependency matrix: $DM_c[n][rid_d] \leftarrow max(DM_c[n][rid_d], ut_d)$. It then hands the read value, v_d, to the caller of GET.

PUT. Client c sends a request $\langle \text{PUT } k, v, DM_c \rangle$ to a partition at the local data center, where k is the item key and v is the update value. Upon receiving the request, p_n^m performs the following steps: 1) Increment $VV_n^m[m]$; 2) Create a new version d for the item identified by k; 3) Assign item key: $k_d \leftarrow k$; 4) Assign item value: $v_d \leftarrow v$; 5) Assign update timestamp: $ut_d \leftarrow VV_n^m[m]$; 6) Assign dependency matrix: $dm_d \leftarrow DM_c$; 7) Assign source replica id: $rid_d \leftarrow m$. These steps form one atomic operation, and none of them is blocking. p_n^m stores d on stable storage and overwrites the existing version if there is one. It then sends a reply $\langle \text{PUTREPLY } utd_d, rid_d \rangle$ back to the client. Upon receiving the reply, the client updates its dependency matrix: $DM_c \leftarrow \mathbf{0}$ (reset all elements to zero) and $DM_c[n][rid_d] \leftarrow ut_d$.

Update Replication. A partition propagates its local updates to its replicas at remote data centers in their update timestamp order. To replicate a newly updated item, d, partition p_n^s sends an update replication request $\langle \text{REPLICATE } k_d, v_d, ut_d, dm_d, rid_d \rangle$ to all other replicas.

A partition also applies updates propagated from other replicas in their update timestamp order. Upon receiving the request, partition p_n^m guarantees causal consistency by performing the following steps:

1) p_n^m checks if it has installed the dependency states of d specified by $dm_d[n]$. p_n^m waits until $VV_n^m \geqslant dm_d[n]$, i.e., $VV_n^m[i] \geqslant dm_d[n][i]$, for $0 \leqslant i \leqslant M-1$.

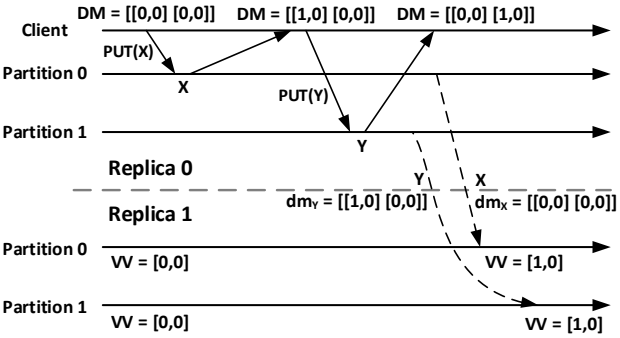

Figure 2: An example of the DM protocol with two partitions replicated at two data centers. A client updates item X and then item Y at two partitions. X and Y are propagated concurrently but their installation at the remote data center is constrained by causality.

2) p_n^m checks if causality is satisfied at the other local partitions. p_n^m waits until, $VV_j^m \geqslant dm_d[j]$, for $0 \leqslant j \leqslant N-1$ and $j \neq n$. It needs to send a message to p_j^m for dependency checking if $dm_d[j]$ contains at least one non-zero element.

3) If there is currently no value stored for item k_d at p_n^m, it simply stores d. If p_n^m has an existing version d' such that $k_{d'} = k_d$, it orders the two versions deterministically by concatenating the update timestamp (high order bits) and source replica id (low order bits). If d is ordered after d', p_n^m overwrites d' with d. Otherwise, d is discarded.

4) p_n^m updates its version vector: $VV_n^m[s] \leftarrow ut_d$. As updates are propagated in order, we have the invariant before $VV_n^m[s]$ is updated: $VV_n^m[s] + 1 = ut_d$.

Example. We give an example to explain the necessity for dependency matrices. Consider a system with two partitions replicated at two data centers ($N = 2$ and $M = 2$) in Figure 2. The client first updates item X at the local data center. The update is handled by partition p_0^0. Upon receiving a response the client updates item Y at partition p_1^0. Causality through the client session dictates that $X \rightsquigarrow Y$. At the other data center, p_1^1 applies Y only after p_0^1 applies X. The figure shows the dependency matrices propagated with the updates. p_1^1 waits until the version vector of p_0^1 is no less than $[1,0]$, which guarantees that X has been replicated.

Correctness. The DM protocol uses dependency matrices to track the nearest dependencies of a client session or an item. It resets the dependency matrix of a client session after each PUT to keep its size after encoding small. This does not affect correctness. By only remembering the update timestamp of the PUT, the protocol utilizes the transitivity of causality to track dependencies correctly.

An element in a dependency matrix, a scalar value,

is the maximum update timestamp of all the nearest dependency items from the corresponding partition. Since a partition always propagates local updates to and applies remote updates from other replicas in their update timestamp order, once it applies an update from another replica, it must have applied all the other updates with a smaller update timestamp from the same replica. Therefore, if a partition satisfies the dependency requirement specified by the dependency matrix of an update, it must have installed all the dependencies of the update.

3.3 Cost over Eventual Consistency

Compared with typical implementations of eventual consistency, the DM protocol introduces some overhead to capture causality. This is a reasonable price to pay for the stronger semantics.

Storage Overhead. The protocol keeps a dependency matrix and other metadata for each item. The per-item dependency metadata is the major storage overhead of causal consistency. We keep it small by only tracking the nearest dependencies and compressing it by sparse matrix encoding. In addition, the protocol requires that a client session maintains a dependency matrix and a partition maintains a version vector. These global states are small and negligible.

Communication Overhead. Checking dependencies of an update during replication requires a partition to send a maximum of $N-1$ messages to other local partitions. If a row in the dependency matrix contains all zeros, then the corresponding partition does not need to be checked since the update does not *directly* depend on any states managed by all replicas of that partition. In addition, the dependency metadata carried by update replication messages also contributes to inter-datacenter network traffic.

3.4 Conflict Detection and Resolution

The above protocol orders updates of the same item deterministically by using the update timestamp and source replica id. If there are no updates for a long enough time, replicas of the same partition eventually converge to the same state while respecting causality. However, the dependency metadata does not indicate whether two updates from different replicas are conflicting or not.

To support conflict detection, we extend the DM protocol by introducing one more member to the existing dependency metadata: the item dependency timestamp. We denote the item dependency timestamp of an item d by idt_d. When d's source partition p_n^s creates d, it assigns to idt_d the update timestamp of the existing version of item k_d if it exists or -1 otherwise. When partition p_n^m applies d after dependency checking, it handles the existing version d' as below. If d is created after d' is replicated at p_n^s, $idt_d = ut_{d'}$, then d and d' do not conflict. p_n^m overwrites d' with d. If d and d' are created concurrently by different replicas, $idt_d \neq ut_{d'}$, they conflict. In that case, either the application can be notified to resolve the conflict using application semantics, or p_n^m orders the two conflicting updates deterministically as outlined in Section 3.2.

4 DM-Clock Protocol

In this section, we describe the DM-Clock protocol, which extends the DM protocol to support causally consistent read-only transactions. Many applications can benefit from a programming interface that provides a causally consistent view on multiple items, as an example later in this section shows.

Compared with the DM protocol, the DM-Clock protocol keeps multiple versions of each item. It also requires accesses to physical clocks. It assigns each item version a physical update timestamp, which imposes on causally related item versions a total order consistent with the (partial) causal order. A read-only transaction obtains its snapshot timestamp by reading the physical clock at the first partition it accesses, its *originating partition*. The DM-Clock protocol then provides a causally consistent snapshot of the data store, including the latest item versions with a physical update timestamp no greater than the transaction's snapshot timestamp.

4.1 Read-only Transaction

With the DM-Clock protocol, the key-value store also provides a transactional read operation:

- $\langle \text{vals} \rangle \leftarrow$ GET-TX($\langle \text{keys} \rangle$): This operation returns the values of a set of items identified by *keys*. The returned values are causally consistent.

A read-only transaction provides a causally consistent snapshot of the data store. Assume x_r and y_r are two versions of items X and Y, respectively. If a read-only transaction reads x_r and y_r, and $x_r \rightsquigarrow y_r$, then there does not exist another version of X, x_o, such that $x_r \rightsquigarrow x_o \rightsquigarrow y_r$.

We give a concrete example to illustrate the application of read-only transactions. Assume Alice wants to share some photos with friends through an online social network such as Facebook. She first changes the permission of an album from "public" to "friends-only" and then uploads some photos to that album. When these two updates are propagated to and applied at remote replicas, causality ensures that their occurrence order is preserved: the permission update operation happens before the photo upload operation. However, it is possible that Bob, not a friend of Alice, first reads the permission of the album as "public" and then sees the photos that were uploaded after the album was changed to "friends-only". Enclosing the album permission check and the viewing

of the photos in a causally consistent read-only transaction prevents this undesirable outcome. In a causally consistent snapshot, the photos cannot be viewed if the permission change that causally precedes their uploads is not observed.

4.2 Definitions

Physical Clocks. The DM-Clock protocol uses loosely synchronized physical clocks. We assume each server is equipped with a hardware clock that increases monotonically. A clock synchronization protocol, such as the Network Time Protocol (NTP) [1], keeps the clock skew under control. The clock synchronization precision does not affect the correctness of our protocol. We use $Clock_n^m$ to denote the current physical clock time at partition p_n^m.

Dependency Metadata. Compared with the DM protocol, the DM-Clock protocol introduces additional dependency metadata. Each version of an item d has a *physical update timestamp*, put_d, which is the physical clock time at the source partition of d when it is created. Different versions of an item are sorted in the item's version chain using their physical update timestamps. A client c maintains a *physical dependency time*, PDT_c, for its session. This variable stores the greatest physical update timestamp of all the states a client session depends on. Each partition p_n^m maintains a *physical version vector*, PVV_n^m, a vector of M physical timestamps. $PVV_n^m[i]$ ($0 \leqslant i \leqslant M-1, i \neq m$) indicates the physical time of p_n^i seen by p_n^m. This value comes either from replicated updates or from heartbeat messages.

4.3 Protocol

We now describe how the DM-Clock protocol extends the DM protocol to support read-only transactions.

GET. When a partition returns the read version d back to the client, it also includes its physical update timestamp put_d. Upon receiving the reply, the client updates its physical dependency time: $PDT_c \leftarrow max(PDT_c, put_d)$.

PUT. An update request from client c to partition p_n^m also includes PDT_c. When p_n^m receives the request, it first checks whether $PDT_c < Clock_n^m$. If not, it delays the update request until the condition holds. When p_n^m creates a new version d for the item identified by k_d, it also assigns d the physical update time: $put_d \leftarrow Clock_n^m$. It then inserts d to the version chain of item k_d using ut_d. The reply message back to the client also includes put_d. Upon receiving the reply message, the client updates its physical dependency time: $PDT_c \leftarrow max(PDT_c, put_d)$.

With the above read and update rules, our protocol provides the following property on causally related states: *For any two item versions x and y, if $x \rightsquigarrow y$, then $put_x < put_y$.*

Update Replication. An update replication request of d also includes put_d. When partition p_n^m receives d from p_n^s and after d's dependencies are satisfied, it inserts d into the version chain of item k_d using put_d. p_n^m then updates its physical version vector: $PVV_n^m[s] \leftarrow put_d$.

Heartbeat Broadcasting. A partition periodically broadcasts its current physical clock time to its replicas at remote data centers. It sends out heartbeat messages and updates in the physical timestamp order.

When p_n^m receives a heartbeat message with physical time pt from p_n^s, it updates its physical version vector: $PVV_n^m[s] \leftarrow pt$. We use Δ to denote the heartbeat broadcasting interval. (Our implementation of Orbe sets Δ to 10ms.) A partition skips sending a heartbeat message to a replica if there was an outgoing update replication message to that replica within the previous Δ time. Hence, heartbeat messages are not needed when replicas exchange updates frequently enough.

GET-TX. A read-only transaction t maintains a physical *snapshot timestamp* st_t and a *readset* rs_t. Client c sends a request $\langle$GETTX *kset*$\rangle$ to a partition by some load balancing algorithm, where *kset* is the set of items to read.

When t is initialized at p_o^m, the originating partition, it reads the local hardware clock to obtain its snapshot timestamp: $st_t \leftarrow Clock_o^m - \Delta$. We provide a slightly older snapshot to a transaction, by subtracting some amount of time from the latest physical clock time, to reduce the probability of a transaction being delayed and the duration of the delay. p_o^m reads the items specified by *kset* one by one. If p_o^m does not store an item required by t, it reads the item from another local partition that stores the item.

Before t reads an item at partition p_n^m, it first waits until two conditions hold: 1) $Clock_n^m \geqslant st_t$; 2) $min(\{PVV_n^m[i] \mid 0 \leq i \leq M-1, i \neq m\}) \geqslant st_t$. t then chooses the latest version d such that $put_d \leqslant st_t$ from the version chain of the read item and adds d to its readset rs_t. After t finishes reading all the requested items, it sends a reply $\langle$GETTXREPLY $rs_t\rangle$ back to the client. The client handles each retrieved item version in rs_t one by one in the same way as for a GET operation.

By delaying a read-only transaction under the above conditions, our protocol achieves the following property: *The snapshot of a transaction includes all item versions with a physical update timestamp no greater than its snapshot timestamp, if no failure happens.*

If failure happens, a read-only transaction may be blocked. We provide solutions to this in Section 5, where we discuss failure handling.

Correctness. To see why our protocol provides causally consistent read-only transactions, consider $x_r \rightsquigarrow y_r$ in the definition in Section 4.1 again. With the first property, if a transaction t reads x_r and y_r, then

$put_{x_r} < put_{y_r} \leq st_t$. With the second property, t only reads the latest version of an item with a physical update timestamp no greater than st_t. Hence there does not exist x_o such that $put_{x_r} < put_{x_o} \leq st_t$. Therefore, it is impossible that there exists x_o such that $x_r \rightsquigarrow x_o$. Our protocol provides causally consistent read-only transactions.

Conflict Detection and Resolution. The above DM-Clock protocol cannot tell whether two updates conflict or not. We employ the same technique used by the DM protocol in Section 3.4 to detect conflicts. We do not need the conflict resolution part here since the DM-Clock protocol uses physical update timestamps to totally order different versions of the same item.

4.4 Garbage Collection

The DM-Clock protocol stores multiple versions of each item. We briefly describe how to garbage-collect old item versions to keep the storage footprint small. Partitions within the same data center periodically exchange snapshot timestamps of the oldest active transactions. If a partition does not have any active read-only transactions, it sends out the latest physical clock time. At each round of garbage collection, a partition chooses the minimum one among the received timestamps as the *safe garbage collection timestamp*. With this timestamp, a partition scans the version chain of each item it stores. It only keeps the latest item version created before the safe garbage collection timestamp (if there is one) and the versions created after the timestamp. It removes all the other versions that are not needed by active and future read-only transactions.

5 Failure Handling

We briefly describe how the DM and DM-Clock protocols handle failures.

5.1 DM Protocol

Client Failures. When a client fails, it stops issuing new requests to the data store. The failure of a client does not affect other clients and the data store. Recovery is not needed since a client only stores soft states for dependency tracking.

Partition Server Failures. A partition maintains a redo log on stable storage, which stores all installed update operations in the update timestamp order. A failed partition recovers by replaying the log. Checkpointing can be used to accelerate the recovery process. The partition then synchronizes its states with other replicas at remote data centers by exchanging locally installed updates.

The current design of the DM protocol does not tolerate partition failures within a data center. However, it can be extended to tolerate failures by replicating each data partition within the same data center using standard techniques, such as primary copy [4, 20], Paxos [16] and chain replication [23].

Data Center Failures. The DM protocol tolerates the failure of an entire data center, for example due to power outage, and network partitions among data centers. If a data center fails and recovers later, it rebuilds the data store states by recovering each partition in parallel. If the network partitions and heals later, it updates the data store states by synchronizing the operation logs within each replication group in parallel. If a data center fails permanently and cannot recover, any updates originated in the failed data center, which are not propagated out, will be lost. This is inevitable due to the nature of causally consistent replication, which allows low-latency local updates without requiring coordination across data centers.

5.2 DM-Clock Protocol

The DM-Clock protocol uses the same failure handling techniques of the DM protocol, except that it treats read-only transactions specially.

With the DM-Clock protocol, before reading an item at a partition, a read-only transaction requires that the partition has executed all local updates and applied all remote updates with update timestamps no greater than the snapshot timestamp of the transaction. However, if a remote replica fails or the network among data centers partitions, a transaction might be delayed for a long time because it does not know whether there are any updates from a remote replica that should be included in its snapshot but have not been propagated.

Two approaches can solve this problem. If a transaction is delayed longer than a certain threshold, its originating partition re-executes it using a smaller snapshot timestamp to avoid blocking on the (presumably) failed or disconnected remote partition. With this approach, the transaction provides relatively stale item versions until the remote partition reconnects. A transaction delayed for enough long time can also switch to a two-round protocol similar to the one used in Eiger [18]. In this case, the transaction returns relatively fresh data but may need two rounds of messages to finish.

6 Dependency Cleaning

In this section, we present *dependency cleaning*, a technique that further reduces the size of dependency metadata in the DM and DM-Clock protocols. This idea is general and can be applied to other causally consistency systems.

6.1 Intuition

Although our DM and DM-Clock protocols effectively reduce the size of dependency metadata by using dependency matrices and a few other techniques, for

big data sets managed by a large number of servers, it is still possible that the dependency matrix of an update has many non-zero elements. For instance, a client may scan a large number of items located at many different partitions and update a single item in some statistics workloads. In this case, the dependency metadata can be many times bigger than the actual application payload, which incurs more inter-datacenter traffic for update replication. In addition, checking dependencies of such an update during replication requires a large number of messages.

The hidden assumption behind tracking dependencies at the client side is that a client does not know whether a state it accesses has been fully replicated by all replicas. To guarantee causal consistency, the client has to remember all the nearest dependency states it accesses and associates them to subsequent update operations. Hence when an update is propagated to a remote replica, the remote replica uses its dependency metadata to check whether all its dependency states are present there. This approach is *pessimistic* because it assumes the dependency states are not replicated by all replicas. Most existing solutions for causal consistency are built on this assumption. This is a valid assumption if the network connecting the replicas fails or disconnects often, which the early works are based upon [21, 22]. However, it is not realistic for modern data center applications. Data centers of the same organization are often connected by high speed, low latency, and reliable fiber links. Most of the time, network partitions among data centers only happen because of accidents, and they are rare. Therefore, we argue that one should not be pessimistic about dependency tracking for this type of applications. If a state and its dependencies are known to be fully replicated by all replicas, a client does not need to include it in the dependency metadata when reading it. With this observation, we can substantially reduce the size of the dependency metadata.

6.2 Protocol Extension

We describe how to extend the DM and DM-Clock protocols to support dependency cleaning. To track updates that are replicated by all replicas, we introduce a *full replication version vector* (RVV) at each partition. At partition p_n^m, RVV_n^m indicates that the first $RVV_n^m[i]$ updates of p_n^i ($0 \leqslant i \leqslant M-1$) have been fully replicated.

Update Replication. We add the following extensions to the update replication process. After p_n^m propagates an item version d to all other replicas, it requires them to send back a *replication acknowledgment* message after they apply d. Similar to propagating local updates in their update timestamp order, a partition also sends replication acknowledgments in the same order. Once p_n^m receives replication acknowledgments of d from all other replicas, it increments $RVV_n^m[m]$. p_n^m then sends a *full-replication completion* message of d to other replicas. Similarly, the full-replication completion messages are also sent in the update timestamp order. Upon partition p_n^i receives the full-replication completion of d, it increments $RVV_n^i[m]$ ($0 \leqslant i \leqslant M-1$ and $i \neq m$). Since partition p_n^m increments an element of VV_n^m when applying a replicated update but increments the corresponding element in RVV_n^m only after that update is fully replicated, $RVV_n^m \leqslant VV_n^m$ always holds.

GET and GET-TX. We now describe how RVV is used to perform dependency cleaning when the data store handles read operations. Assume a client sends a read request to partition p_n^m and an item version d is selected. If $RVV_n^m[rid_d] \geqslant ut_d$, p_n^m knows that d and all its dependency states have been fully replicated and there is no need to include d in the dependencies of the client session. p_n^m sends a reply message back to the client without including d's update timestamp. Upon receiving the reply, the client keeps its dependency matrix unchanged. This technique can also be applied to read-only transactions similarly. Therefore, by marking an item version as fully replicated, this technique "cleans" the dependency it introduces to the client that reads it.

Normally, the duration that $RVV_n^m[rid_d] < ut_d$ holds is short. Under moderate system loads, this duration is roughly 1.5 WAN round-trip latency plus two times the write latency of stable storage , which is normally a few hundreds milliseconds. As a consequence, for a broad spectrum of applications, most read operations do not generate dependencies, which keeps the dependency metadata small.

6.3 Message Overhead

Dependency cleaning has a tradeoff. It reduces the size of dependency metadata and the cost of dependency checking at the expense of more network messages for sending replication acknowledgments and full-replication completions.

Assume an update depends on states from k partitions except its source partition. For a system with M replicas, without dependency cleaning, it takes $M-1$ WAN messages to propagate the update to all other replicas at remote data centers. The dependency matrix in each of the $M-1$ WAN messages has at least k non-zero rows. $(M-1)k$ LAN messages are required for checking dependencies of the propagated update at remote replicas. With dependency cleaning, it requires $3(M-1)$ WAN messages to replicate an update. For many workloads, the dependency matrix of an update contains mostly zero elements. Almost no LAN messages are needed for dependency checking.

7 Evaluation

We evaluate the DM protocol, DM-Clock protocol, and dependency cleaning in Orbe, a multiversion key-value store that supports both partitioning and replication. In particular, we answer the following questions:

- Does Orbe scale as the number of partitions increases?
- What is the overhead of providing causal consistency compared with eventual consistency?
- How does Orbe compare with COPS?
- Is dependency cleaning an effective technique for reducing the size of dependency metadata?

7.1 Implementation and Setup

We implement Orbe in C++ and use Google's Protocol Buffers for message serialization. We partition the data set to a group of servers using consistent hashing [12]. We run NTP to keep physical clocks synchronized. NTP can be configured to change the clock frequency to catch up or fall back to a target. Hence, physical clocks always move forward during synchronization, a requirement for correctness in Orbe.

As part of the application tier, servers that run Orbe clients are in the same data center with Orbe partition servers. A client chooses a partition in its local data center as its originating partition by a load balancing scheme. The client then issues all its operations to its originating partition. If the originating partition does not store an item required by a client request, it executes the operation at the local partition that manages the required item.

Orbe's underlying key-value store keeps all key-value pairs in main memory. A key points to a linked list that contains different versions of the same item. The operation log resides on disk. The system performs group commit to write multiple updates in one disk write. A PUT operation inserts a new version to the version chain of the updated item and adds a record to the operation log. During replication, replicas of the same partition exchange their operation logs. Each replica replays the log from other replicas and applies the updates one by one after dependency checking.

We run the DM-Clock protocol in all experiments, even where the DM protocol would suffice, because it is a superset of the DM protocol. We set the heartbeat broadcasting interval Δ to 10ms. By default, dependency cleaning is disabled. We enable it in one experiment, where we mention its use explicitly (see Section 7.6).

We run experiments on a local cluster where all servers are connected by a single GigE switch. All servers in the cluster are Dell PowerEdge SC1425 running Linux 3.2.0. Each server has two Intel Xeon processors, 4GB of DDR2 memory, one 7200rpm 160GB SATA disk, and one GigE network port. The round-trip network latency in our local cluster is between 120 to 180 microseconds. We enable the hardware cache of the disk. The latency of writing a small amount of data (64B) to the disk is around 450 microsecond. We partition the local cluster into multiple logical "data centers" as necessary. We introduce an additional 120 milliseconds network latency for messages among replicas of the same partition located at different logical data centers.

Operation	Echo	GET-10B	PUT-1B	PUT-16B	PUT-128B
Throughput (K op/s)	71.3	61.4	36.8	36.4	30.2

Table 2: Maximum throughput of client operations on a single partition server without replication.

7.2 Microbenchmarks

We first evaluate the basic performance characteristics of Orbe through microbenchmarks. We replicate the data set in two data centers. At each data center, the data set is partitioned on eight servers. Each partition loads one million data items during initialization. For each preloaded item, the size of a key is eight bytes and the value is ten bytes.

In the first experiment, we examine the capability of a single partition server. We launch enough clients to saturate the server. A GET operation reads a randomly selected item from its originating partition. The PUT operation also operates on the originating partition by updating a random item with different sizes of update values. For comparison, we also introduce an Echo operation, which simply returns the operation argument to the clients.

As shown in Table 2, a partition server can process Echo operations at about 70K ops/s, GET operations at about 60K ops/s, and PUT operations at about 30K ops/s. The throughput of Echo indicates the message processing capability of our hardware. As the update value size increases in PUT, the throughput drops slightly due to the increased cost of memory copies. In all cases, CPU is the bottleneck.

In the second experiment, we measure operation latencies. For this experiment, GET and PUT choose items located at the originating partition with a probability of 50% and at other local partitions with the other 50%. A GET-TX operation reads six items in total. One is from its originating partition while the other five are from other local partitions.

Figure 3 shows the latency distribution of the four operations. The Echo operation shows the baseline as it

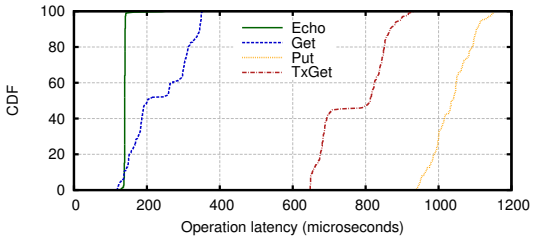

Figure 3: Latency distribution of client operations.

only takes one round-trip latency to finish. Each GET and PUT requires either one or two rounds of messages within the same data center (depending on the location of the requested item), which results in two clear groups of latencies. The latency of executing a client operation is low, because Orbe does not require a partition server to coordinate with its replicas at other data centers to process GET and PUT operations. None of the (microsecond-scale) client operations depend on replication operations that incur the 120ms latency between data centers.

7.3 Scalability

We now examine the scalability of Orbe with an increasing number of partitions. We set up two data centers with two to eight partitions at each.

We first use three workloads which are a configurable mix of PUTs and GETs. Items are selected from the originating partition with a probability of 50% and from other local partitions for the other 50%. PUT operations update items with ten bytes values. For the workload of read-only transactions, each GET-TX reads one item from its originating partition and five from other local partitions. Figure 4 shows the throughput of Orbe as the number of partitions increases. Regardless of the put:get ratio, Orbe scales out with an increasing number of partitions. Because Orbe propagates updates across partitions in parallel, it is able to utilize more servers to provide higher throughput.

7.4 Comparison with Eventual Consistency

To show the overhead of providing causal consistency in our protocols, we compare Orbe with an eventually consistent key-value store, which is implemented in Orbe's codebase. We set up two data centers of three partitions each. A client accesses items randomly selected from the three local partitions with different put:get ratios. PUT updates an item with a value of 60 bytes.

Figure 5 shows the throughput of Orbe and the eventually consistent key-value store. For an almost read-only workload, they have similar throughputs. For an almost update-only workload, Orbe's throughput is about 24% lower. The minor degradation in throughput is a reasonable price to pay for the much improved semantics over eventual consistency.

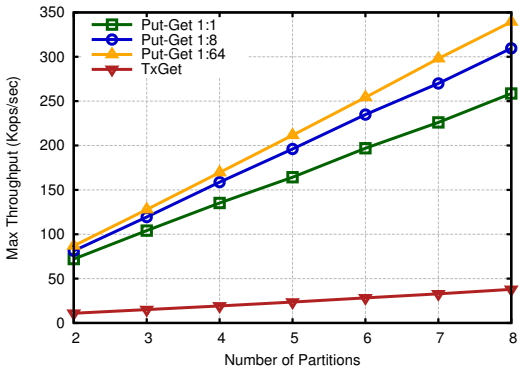

Figure 4: Maximum throughput of varied workloads with two to eight partitions. The legend gives the put:get ratio.

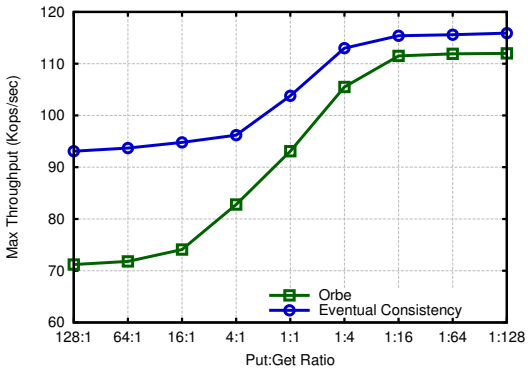

Figure 5: Maximum throughput of workloads with varied put:get ratios for both Orbe (causal consistency) and eventual consistency.

The major overhead of implementing causal consistency comes from 1) network messages for dependency checking and 2) processing, storing and transmitting the dependency metadata. Figure 6 shows this overhead with two curves. The first is the average number of dependency checking messages per replicated update. The second is the percentage of the dependency metadata in the update replication traffic in Orbe. When the workload is almost update-only, the metadata percentage is small and so is the number of dependency checking messages per replicated update. When the workload becomes read-heavy, the numbers go up, but level off after GETs dominate the workload.

The contents of dependency matrices explain the numbers in Figure 6. As Orbe tracks only the nearest dependencies, an update depends only on the previous update and the reads since the previous update in the same client session. With a high put:get ratio, the dependency matrix contains only a few non-zero elements. With a low put:get ratio, reads generate a large number

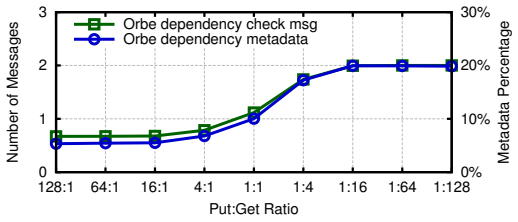

Figure 6: Average number of dependency checking messages per replicated update and percentage of dependency metadata in the update replication traffic with varied put:get ratios in Orbe.

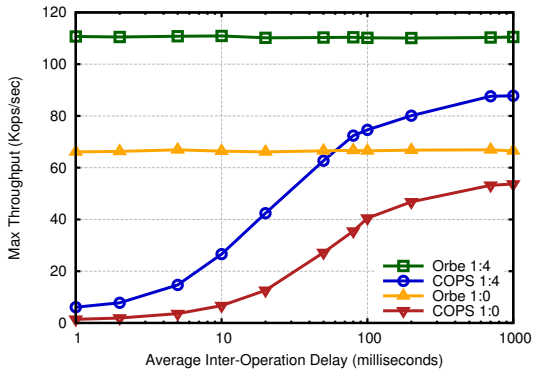

Figure 7: Maximum throughput of operations with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

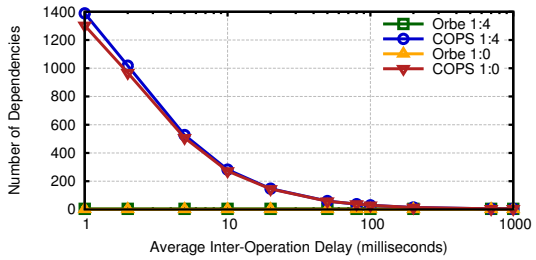

Figure 8: Average number of dependencies for each PUT operation with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

of dependencies, but the total number of elements in a dependency matrix is bounded by the number of partition servers in the data store.

7.5 Comparison with COPS

We compare Orbe with COPS [17], which also provides causal consistency for both partitioned and replicated data stores. We implement COPS in Orbe's codebase. For an apples-to-apples comparison, we enable read-only transaction support in both Orbe and COPS (which is called COPS-GT in prior work [17]).

COPS explicitly tracks each item version read and updated at the client side. It associates a set of dependency item versions with each update. A dependency matrix in Orbe plays the same role as a set of dependency item versions in COPS, but most of the time it takes less space using sparse encoding.

Although COPS relies on a number of techniques to reduce the size of dependency metadata, it can still become considerable since COPS has to track the *complete* dependencies to support read-only transactions while Orbe only tracks the nearest dependencies. In addition, the execution time of the two-round transactional reading protocol in COPS limits the frequency of garbage-collecting the dependency metadata. During the fixed interval between two successive garbage collections, the more operations a client issues, the more dependency states it creates. Hence, the size of dependency metadata is highly related to the inter-operation delays at each client. COPS sets the garbage collection interval to six seconds [17]. We use the same value in our COPS implementation.

We set up two data centers of three partitions each. A client accesses data items randomly selected from the three partitions with a configurable put:get ratio. Figure 7 illustrates the throughput of Orbe and COPS with different client inter-operation delays. Figure 8 shows the average number of states on which an update depends. For COPS, this is the number of dependency item versions. For Orbe, this is the number of non-zero elements in the dependency matrix. Orbe provides consistently higher throughput than COPS as it tracks fewer dependency states and spends fewer CPU cycles on message serialization and transmission. Figure 8 suggests that Orbe and COPS should have similar throughput when the inter-operation delays are longer than one second.

By tracking fewer dependency states and efficiently encoding the dependency matrix, Orbe also reduces the inter-datacenter network traffic for update replication among replicas of the same partition. Figure 9 compares the aggregated replication traffic (transmission only) between Orbe and COPS.

Tracking fewer states at the client side reduces the client's memory footprint, but also consumes fewer CPU cycles as fewer temporary objects are created and destroyed for tracking dependencies. Figure 10 shows the CPU utilization of a server that runs a group of clients for Orbe and COPS, separately. For this measurement, we run all clients at a single powerful server to saturate the data store and record the CPU utilization of the server. Orbe is more efficient. It uses fewer CPU cycles per operation as it manages fewer states.

7.6 Dependency Cleaning

Dependency cleaning removes the necessity for dependency tracking when a client reads a fully replicated

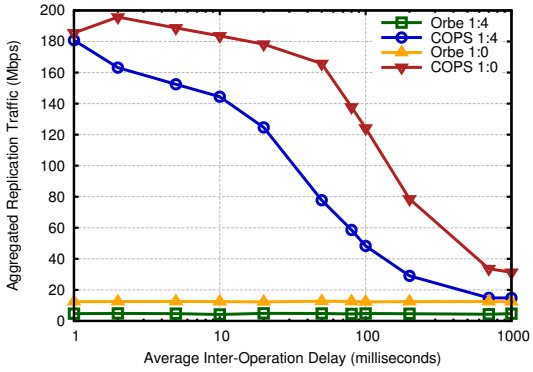

Figure 9: Aggregated replication transmission traffic across data centers with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

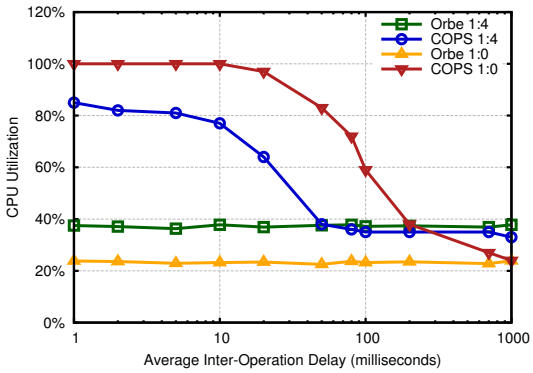

Figure 10: CPU utilization of a server that runs a group of clients with varied inter-operation delays for both Orbe and COPS. The legend gives the put:get ratio.

item version. However, it increases the cost of update replication as it requires additional inter-datacenter messages to mark an item version as fully replicated.

In this experiment, we show the benefits of dependency cleaning for workloads that read from a large number of partitions and update only a few. We set up two data centers and vary the number of partitions from one to eight at each data center. A client reads a randomly selected item from each of the local partitions and updates one random item at its originating partition.

Figure 11 shows the maximum throughput of Orbe with and without dependency cleaning enabled. When the system has only one partition, all reads and updates go to that partition. Dependency cleaning does not help as no network message is required for dependency checking. In this case, the throughput of Orbe with dependency cleaning enabled is slightly lower, because it requires more messages for update replication. However, the throughput drop is small, because we let update replication messages piggyback replication acknowledgement and full-replication completion

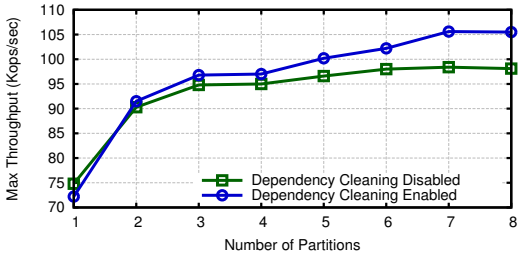

Figure 11: Maximum throughput of operations with and without dependency cleaning enabled.

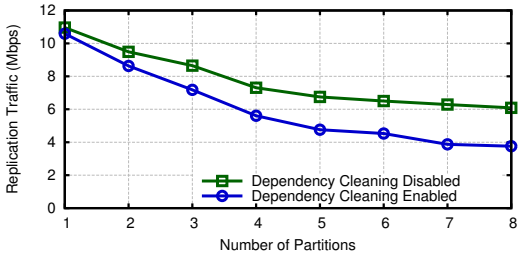

Figure 12: Aggregated replication transmission traffic with and without dependency cleaning enabled.

messages and batch these messages whenever possible.

As we increase the number of partitions, the throughput of Orbe with dependency cleaning enabled is higher. The throughput gap increases as the system has more partitions. With dependency cleaning, an update does not depend on states from other partitions most of the time, although it reads states from those partitions. As a result, dependency checking on replicated updates does not incur network messages at the remote data center. By removing this part of the overhead, dependency cleaning helps the overall throughput.

Dependency cleaning also reduces the size of dependency metadata for an update. Figure 12 shows the aggregated replication traffic from all partitions. The traffic decreases as the number of partitions increases because the put:get ratio decreases. Figure 13 shows the average percentage of metadata in the update replication traffic.

8 Related Work

There have been many causally consistent systems in the literature, such as lazy replication [13], Bayou [22], and WinFS [19]. They use various techniques derived from the causal memory algorithm [2]. However, these systems target only full replication. None of them considers scalable causal consistency for partitioned and replicated data stores, to which Orbe provides a solution.

COPS [17] identifies the problem of causal consistency for both partitioned and replicated data stores and gives a solution. It tracks every accessed state as dependency metadata at the client side. To support causally

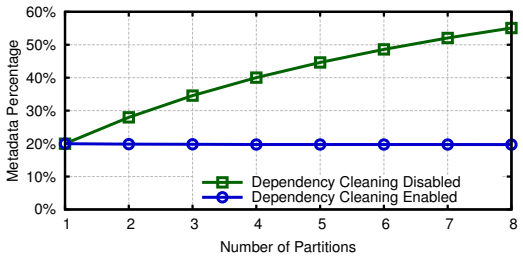

Figure 13: Average percentage of dependency metadata in update replication traffic with and without dependency cleaning enabled.

consistent read-only transactions, a client has to track the complete dependencies explicitly. Although COPS garbage-collects the dependency metadata periodically, the metadata size may still be large under many workloads and can affect performance. In comparison, Orbe relies on dependency matrices to track the nearest dependencies and keeps the dependency metadata small and bounded. Orbe provides causally consistent read-only transactions using loosely synchronized clocks. Orbe requires one round of messages to execute a read-only transaction in the failure-free mode while COPS requires maximum two rounds.

Eiger [18] is recent follow-up work on COPS and provides causal consistency for a distributed column store. It proposes a new protocol for read-only transactions using Lamport clock [14]. Although Eiger also needs maximum two rounds of messages to execute a read-only transaction, a client tracks only the nearest dependencies. In addition, Eiger provides causally consistent update-only transactions. Eiger still tracks every accessed state at the client side.

ChainReaction [3] implements causal consistency on top of chain replication [23]. ChainReaction also tracks every accessed state at the client side. To solve the problem of large dependency metadata when providing read-only transactions, it uses a global sequencer service at each data center to totally order update operations and read-only transactions. However, the sequencer service increases the latency of all update operations by one round-trip network latency within the data center and is a potential performance bottleneck. In comparison, Orbe does not have any centralized component. It provides causal consistency for update-anywhere replication and relies on loosely synchronized physical clocks to implement read-only transactions.

Bolt-on causal consistency [5] provides causal consistency to existing eventually consistent data stores. It inserts a shim-layer between the data store and the application layer to insure the safety properties of causal consistency. It relies on the application to maintain explicit causality relationships. Tracking causality based on application semantics is precise but requires the application developers to specify causal relationships among application operations. In contrast, Orbe provides causal consistency directly in the data store, without requiring coordination from the application.

Physical clocks have been used in many distributed systems. For example, Spanner [8] implements serializable transactions in a geographically replicated and partitioned data store. It provides external consistency [11] based on synchronized clocks with bounded uncertainty, called TrueTime, requiring access to GPS and atomic clocks. Clock-SI [9] uses loosely synchronized clocks to provide snapshot isolation [6] to transactions in a purely partitioned data store. In comparison, Orbe targets causal consistency, a weaker consistency model. It provides causally consistent read-only transactions using loosely synchronized clocks in a partitioned and replicated data store.

9 Conclusion

In this paper, we propose two scalable protocols that efficiently provide causal consistency for partitioned and replicated data stores. The DM protocol extends version vectors to two-dimensional dependency matrices and relies on the transitivity of causality to keep dependency metadata small and bounded. The DM-Clock protocol relies on loosely synchronized physical clocks to provide causally consistent read-only transactions. We implement the two protocols in a distributed key-value store. We show that they incur relatively small overhead for tracking causal dependencies and outperform a prior approach based on explicit dependency tracking.

Acknowledgments. We thank the SOCC program committee and especially our shepherd, Indranil Gupta, for their constructive feedback. Daniele Sciascia also provided useful comments on final revisions of this work.

References

[1] The network time protocol. `http://www.ntp.org`, 2013.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2013.

[4] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on*

Software Engineering, pages 562–570. IEEE Computer Society Press, 1976.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772. ACM, 2013.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM, 1995.

[7] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM symposium on Principles of Distributed Computing*, pages 7–. ACM, 2000.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 251–264. USENIX Association, 2012.

[9] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*. IEEE, 2013.

[10] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[11] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[13] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 100(9):690–691, 1979.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 313–328. USENIX Association, 2013.

[19] D. Malkhi and D. Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):339–353, 2005.

[20] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing*, pages 8–17. ACM, 1988.

[21] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, (3):240–247, 1983.

[22] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 288–301. ACM, 1997.

[23] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design*, pages 91–104, 2004.

[24] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.